8th International Conference on
Combinatorics, Cryptography, Computer Science and Computation
November. 15-16, 2023

Enhancing Code Privacy: An Exploratory Implementation of Java DataFlow Obfuscation

Mohammadhadi Alaeiyan, Ali Hamedi, M.A. SoltanBeigi, AmirReza Parvahan, AmirAli Ghaedi
Faculty of Computer Engineering, K. N. Toosi University of Technology, Seyed Khandan, Shariati Ave, 16317-14191 Tehran, Iran

M.Alaeiyan@kntu.ac.ir, S.Hamedi@email.kntu.ac.ir, M.Soltanbeigi@email.kntu.ac.ir, A.Parvahan@email.kntu.ac.ir, A.Ghaedi@email.kntu.ac.ir

ABSTRACT

In the rapidly evolving landscape of software development, ensuring the security and confidentiality of Java code has emerged as a paramount concern. As the digital realm continues to grow in complexity, so do the threats that target sensitive data and intellectual property. In this context, Java DataFlow Obfuscation stands as a formidable line of defense, harnessing the combined power of ANTLR4 and Python to fortify the security of your Java codebase.

The essay unfolds in distinct phases, commencing with extracting entities and their properties, coupled with subtle code alterations. The renaming process follows, entailing entity encryption, data type transformation, and identifier modifications. The encryption phase comprises the encryption of all entities, data type adjustments, and dynamic parsing upon usage.

The final output is a codebase where entities bear modified identifiers and commodities with primitive data types are encrypted, stringified, and parsed when required. This comprehensive obfuscation strategy enhances code security, and baffles reverse engineering attempts.

In related works, notable contributions in data obfuscation and security are examined, shedding light on the significance of data type manipulation, matrix splitting, and component shuffling within code security.

The conclusion anticipates future directions, envisioning the integration of diverse encryption and decryption algorithms, enhanced compatibility with Java versions and libraries, and optimizations in runtime. These prospects aim to bolster code security further, setting the stage for seamless transitions into code flow obfuscation and abstract interpretation, marking the following milestones in safeguarding Java code.

KEYWORDS: DataFlow Obfuscation, Java Code Security, Entity Encryption, ANTLR4, Data Type Manipulation

1 INTRODUCTION

Ensuring the security and confidentiality of code has perpetually remained a paramount concern for programmers and major corporations. In response to this critical need, we delve into the intricate world of Java [1] DataFlow Obfuscation, leveraging the power of sophisticated tools like ANTLR4 [2] and encryption to fortify code security.

At its core, 'DataFlow' in computer programming governs how data, such as variables, objects, and information, moves within a software system. It influences not just the execution order of statements (control flow) but also how data values are produced and consumed by different parts of the code (data dependency). This concept is vital for ensuring software reliability, efficiency, and security, particularly in languages like Java.

This essay introduces Java DataFlow Obfuscation, a method enhancing code security through systematic code transformations, modified identifiers, and encryption, fortifying defense against reverse engineering and unauthorized access.

ANTLR4, a robust parsing tool, is the engine behind the extraction and parsing of Java code. It allows for precisely identifying entities, data types, and their interconnections. Additionally, encryption techniques are employed to secure sensitive data within the codebase.

In the following sections, we will explore Java DataFlow Obfuscation comprehensively, examining its relevance, intricacies, and real-world applications. The aim is to not only shed light on the importance of dataflow, ANTLR4, obfuscation, and encryption but also to provide a detailed understanding of the methodologies employed to enhance code security in the ever-evolving field of software development.

2 RELATED WORKS

D.I.George Amalarethinam et al. [3] proposed a new and better way for Data Security Enhancement in Public Cloud Storage using Data Obfuscation and Steganography to protect the data in cloud storage. They use MRADO to classically obfuscate (e.g., substitution, redaction or nulling, shuffling, and blurring) the data. After that, they use Pixel Processing using Least Significant Bit (LSB) to hide the data in cover images.

Khaled M. Khan et al. [4] Research focusing on ensuring confidentiality in cloud computing through the innovative methods of matrix splitting, component shuffling, and the incorporation of random noise into matrices is a noteworthy contribution within the field. Exploring related studies in several key areas is imperative to contextualize this work. Firstly, examining the broader landscape of confidentiality in cloud computing elucidates the challenges and existing solutions. Moreover, a detailed investigation of matrix splitting techniques, component shuffling methodologies, and the addition of random noise to matrices is crucial to appreciate the intricacies of these techniques.

Hsiang-Yang Chen et al. [5] As a defense against reverse engineering, introduced this method and designed a changing data type obfuscation method in Java software. They mainly change the variable type from short-term to long-term and long-term to short-term. They do this by connecting two short-term to get a long-term or splitting a long-term into two short-term variables.

Shenoda Guirguis et al. [6] turned standard obfuscation into real-time obfuscations because they must be fast and secure while transmitting data in banking transactions. The data in use is more vulnerable than the data at rest. They did it by utilizing different obfuscation functions for different data types to securely obfuscate the data, for example, by Anonymization.

D.I. George Amalarethinam et al. [7] conducted a comparative analysis of obfuscation techniques, namely WMRADO, MRADO, and MONcrypt. Their research culminated in the proposal of the WMRADO technique, which stands out by employing word-by-word obfuscation as opposed to the line-by-line approach used in MRADO. Furthermore, it is essential to note that when users entrust their data to cloud services, it exposes vulnerabilities not only to attacks on data in transit but also to attacks on data at rest.

3 THE APPROACH

Using ANTLR4 in conjunction with the Java language grammar obtained from the ANTLR grammar GitHub repository [8], we embark on a multi-step process. Our first step involves generating a

Parser, Lexer, and ParserListener, enabling us to extract a semantic representation from raw Java code in text form. To initiate the parsing process, we employ ANTLR's Lexer to tokenize the Java code, which has already been converted into a stream.

Following the successful tokenization, our next pivotal phase commences, entailing the utilization of ANTLR's provided parser. This parser is instrumental in constructing the Java code's Abstract Syntax Tree (AST). Access to the AST is established through the "Compilation Unit" node defined by ANTLR.

The subsequent stage involves a comprehensive traversal of the AST. At this juncture, the core operations of extraction, encryption, and renaming are executed, demonstrating the profound implications of data flow obfuscation.

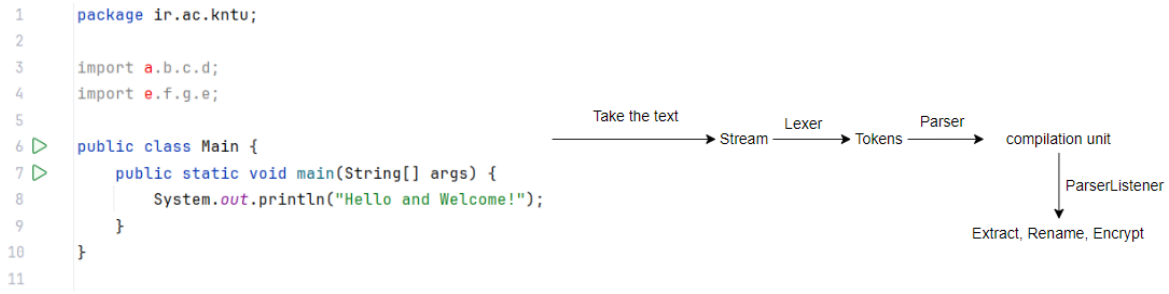


Figure 1. Initial steps before parsing

Having gained access to the Abstract Syntax Tree (AST) derived from the Java code, we are presented with a tree structure that follows the following organization:

1. Compilation Unit: This serves as the primary and foremost unit of the AST, encompassing several essential components, including the package declaration, import declaration, type declaration, and EOF (End of File). In the context of ANTLR, this unit is pivotal in structuring generated code for parsing and language recognition.

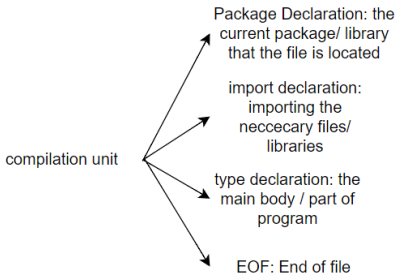


Figure 2. Compilation Unit rule diagram

2. Package Declaration: A package declaration is pivotal in organizing related classes and interfaces within the Java code. In ANTLR, it serves as a structural element for generating code that facilitates parsing and the recognition of language constructs.

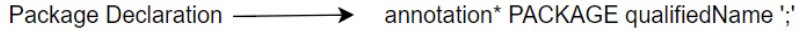


Figure 3. Example 1: a grammar rule derived from ANTLR parser

3. Import Declaration: Import declarations, in the context of Java, enable the inclusion of classes or packages from other namespaces into the code. Within ANTLR, these declarations facilitate the incorporation of grammar rules or Lexer tokens from other ANTLR grammars, enriching the capabilities of the language recognition process.

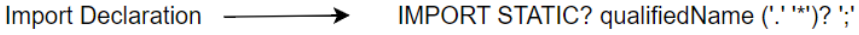


Figure 4. Example 2: a grammar rule derived from ANTLR parser

4. Type Declaration: Type declarations define classes, interfaces, or custom data types in the Java language. Within the domain of ANTLR, these declarations are responsible for specifying Lexer and parser rules, enabling the recognition of language constructs, and contributing to the overall functionality of the parsing process.

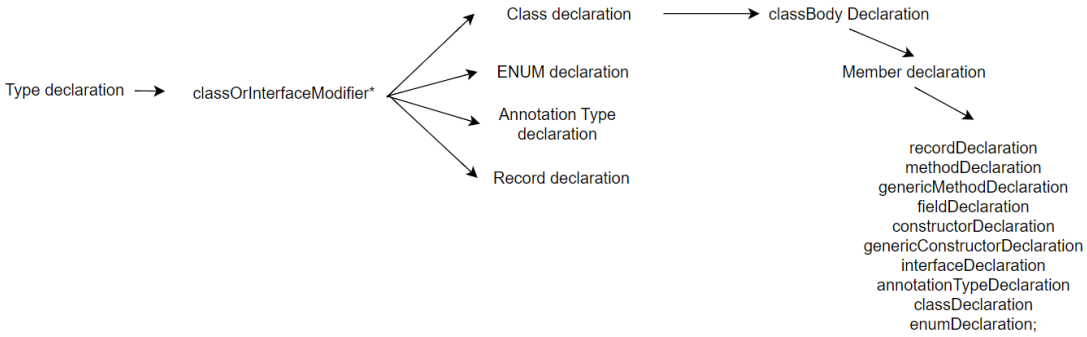


Figure 5. AST representation derived from the extracted grammar rules obtained from ANTLR parser

Subsequently, our process involves a triple-pass parsing of the code, where we systematically undertake the tasks of extraction, renaming, and encryption. Throughout this process, each rule or unit is obtained from the ParserListener, such as CompilationUnit, TypeDeclaration, MemberDeclaration, Variable or Method Declarations, etc.is equipped with an enter and exit function. These functions serve as boundaries, signifying the commencement and conclusion of each specific rule. Furthermore, they provide a context ctx [1] for every rule within the ParserListener, each with a distinct context, such as IdentifierContext, StatementContext, ExpressionContext, and more. These rules follow a hierarchical structure, enabling the systematic and precise manipulation of the code.

It's important to note that for each file within any input project, a token stream is created, and all modifications and code transformations will be carried out within this token stream in the upcoming phases

[1] ANTLR "ctx" refers to the context object used to track and manipulate parse tree nodes in the ANTLR parser generator.

of our obfuscation process. This token stream serves as the basis for reading and subsequent modification of the code within the file and, as a structured representation of lexical elements, becomes the focal point for systematic alterations and enhancements.

3.1 Noise generation

Within our project, we introduce the notion of "noise-code" as a fusion of random variables and corresponding methods meticulously designed to obscure the visual characteristics and underlying intent of code from external observers. Our methodology unfolds in two fundamental steps:

3.1.1. Establishing the Noise-Code Network

This critical phase is orchestrated through the creation of a "Generator" object. The Generator is entrusted with the task of generating random noise-code while systematically categorizing its elements by type and establishing connections between them. The uniqueness of each instance of noise-code is achieved through a deliberate reliance on randomness. The fundamental components of noise-code include:

Noise-Var: These simulated variables are akin to actual variables but possess randomly generated names and types. Importantly, these types must belong to a predefined list of accepted primitives (e.g., Integer, String, Char, Boolean). The number of Noise-Vars is also randomized, resulting in a unique set for each file.

Noise-Method: To inject a layer of meaningful functionality into Noise-Vars, we leverage a curated repository of pre-made Java methods known as the "FuncBank". This repository contains methods designed to match every pair combination of accepted Noise-Var types, allowing us to randomly select a method for a pair of Noise-Vars. The addition of more methods to FuncBank contributes to the unpredictability of the resultant Noise-Code.

These Noise-Methods represent an essential bridge between the abstracted Noise-Vars and the actual code, providing a layer of meaning and functionality within the generated Noise-Code. However, it's not enough for these Noise-Methods to exist as static definitions; they need to actively participate in the code's execution.

To achieve this, our Generator dynamically generates method calls for each Noise-Method and inserts them into the list of Noise-Vars, ensuring that every Noise-Method is invoked during the program's execution.

These code snippets represent our program's original code, carefully crafted to maintain its functionality intact while serving our obfuscation objectives.

3.1.2. Insertion

Our insertion process is guided by a set of algorithms aimed at seamlessly integrating Noise-Code into the existing codebase without easy detection. This approach hinges on dividing the output from our Generator object into multiple groups, each adhering to a crucial rule: The necessary input Noise-Vars for a Noise-Method are always declared and initialized before the method call. The remaining variables in each group are then filled with unused Noise-Vars. Each group is strategically placed within the codebase, ensuring that their inclusion does not lead to syntax errors or raise suspicions.

Moreover, our insertion process goes a step further by strategically placing each group within the existing codebase. This placement is a result of a meticulous analysis of the original code, which enables us to identify optimal positions where our Noise-Code blends seamlessly with the surroundings.

This intelligent design choice not only enhances the authenticity of our obfuscated code but also significantly raises the complexity of reverse engineering attempts. When an observer encounters the Noise-Code, they are met with a structure that follows established coding conventions, reinforcing the illusion of functionality and purpose.

3.2 Extraction

This process is defined as a tool that revolves around the generation of a comprehensive table that encompasses a wide collection of declared entities[2], ranging from variables and methods to classes, interfaces, ENUMs, and more, all of which may necessitate subsequent modification. Within this table, we meticulously observe and catalog their attributes, collecting all pertinent information that holds the potential for future modifications. This systematic approach serves as the cornerstone of our obfuscation strategy by allowing us to implement obfuscation techniques with precision and accuracy.

The structure of this table is elegantly simple. Each row within it represents an individual entity, while each column corresponds to a specific property of that entity. We gather this data during the parsing process facilitated by our Extractor class, an instance of the JavaParserListener[3] class provided by ANTLR.

```
public class TestClass {
    public static void main(String[] args) {

        String testStr = "Test";
        int testInt = 1;
    }
}
```

Figure 6. Example Java code as an input of Extraction process

Table 1. Extracted data table from the code provided in figure 6

Name	Value	FileName	Line	type	VarOrReturnType
TestClass	-	TestClass.java	3	ClassDeclaration	TestClass
main	-	TestClass.java	4	MethodDeclaration	void
testStr	“test”	TestClass.java	6	VariableDeclaration	String
testInt	1	TestClass.java	7	VariableDeclaration	int

[2] In ANTLR, an 'entity' typically refers to a named element in the grammar, representing a distinct rule or token.

[3] In ANTLR, a 'JavaParserListener' is a listener interface generated for a parser, providing methods to react to events during the parsing of a Java source file.

The information captured in this table serves as a valuable resource, offering utility and, at times, a necessary toolkit tailored to the unique requirements of our project. We acquire this information through a two-step process:

3.2.1 Capturing the identifiers

In this step, we leverage the 'enter identifier' method, which captures identifiers following the identifier rule context [4]provided by ANTLR. The context within the 'enterIdentifier' method equips us with essential data for constructing the data scheme to be inserted into our table. The primary data extracted from the identifier context includes the identifier's name, the file it's located in, the line number, and its grammatical rule, referred to as its "type" hereafter. It's important to note that our table includes only identifiers representing accepted types of declarations, excluding calls and primary uses. One of the table's vital columns is 'VarOrReturnType,' which returns different values based on the identifier's type. For variable-like types such as class fields, it returns their normal type, while for methods, it returns their return type.

Following this data extraction, we proceed with some minor code and table modifications. The table modifications involve refining the entities according to our predefined policy to ensure that only entities to be modified are captured. Additionally, we generate random names paired with the identifier names for later use in the renaming steps.

3.2.2 Resolve the needed data for the table

Following the previous step, we progress to complete our data scheme for each individual identifier. This is achieved by leveraging the exit function for each specific targeted rule and the hierarchical structure provided by the JavaParserListener in ANTLR4. The hierarchical structure corresponds to the Java language grammar, where each rule can encompass multiple rules within itself. The parserListener's walk method traverses these rules, utilizing specific events such as "enterParentRule," "enterChildRule," "exitChildRule," and "exitParentRule."

In the illustration presented in Figure 7, you can observe that the identifier rule is a basic rule without any nested rules. This characteristic is crucial to our process. After initially capturing the essential data of the identifier during the "enterIdentifier" event, we proceed to acquire the remaining properties of the identifier within the "exit" method of its corresponding rule. This data includes vital information for our program, such as "varOrReturnType," which indicates whether the identifier represents a variable or a class, and its inheritance status, particularly when it's related to a class. With this essential data at our disposal, we can seamlessly proceed with the renaming phase of the obfuscation process.

[4] in ANTLR refers to the contextual object associated with a parsing rule, holding crucial information about the current parsing state.

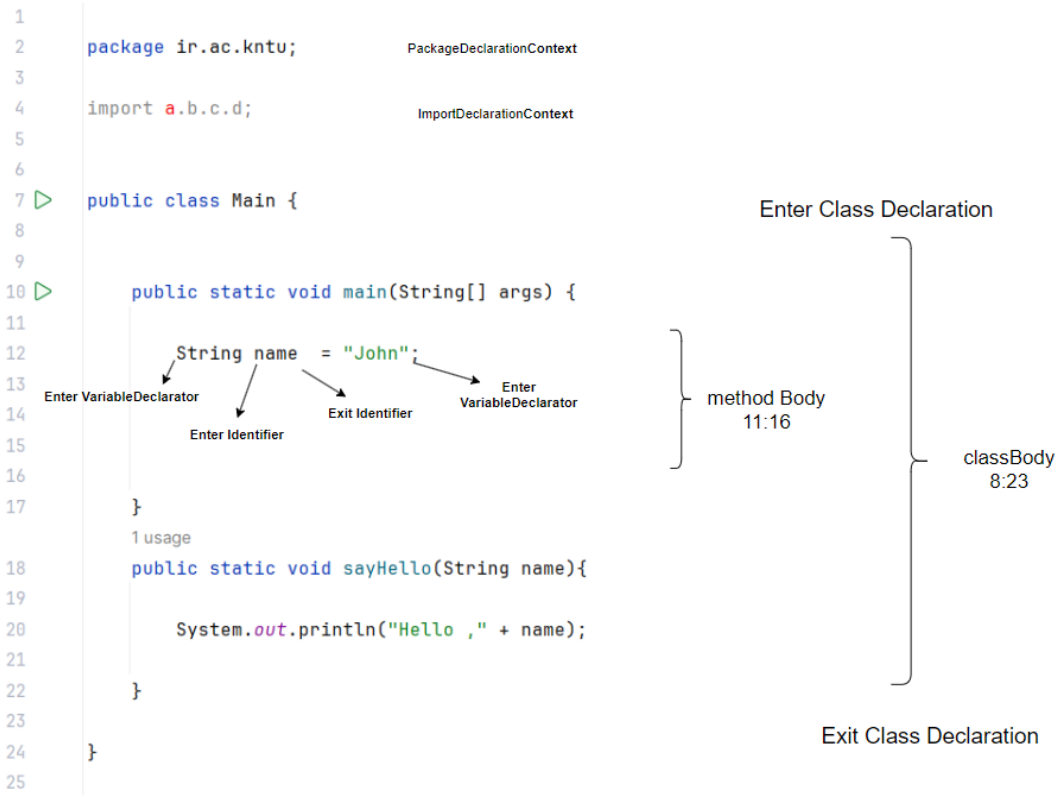


Figure 7. The hierarchical structure derived from the Java code by the grammar

3.3 Renaming

Renaming stands as one of the pivotal pillars of dataflow obfuscation, substantially diminishing readability and severing the semantic link between variable names, their assigned values, and their intended purposes. The renaming step emerges as a conceptual core within our obfuscation process, replete with its own unique challenges and a general solution. We outline a selection of the main challenges we encounter, including:

1. Ensuring precise pairing of the identifier with the correct entity from our table
2. Implementing distinct renaming mechanisms for both identifiers and typeIdentifiers (including classes, ENUMs, interfaces, and records)
3. Adhering to our established renaming policy while exercising discretion to avoid changes in unnecessary identifiers
4. Resolving the complexities of identifier ownership

We address these challenges through the following steps:

3.3.1 Capturing all the identifiers and resolving the identifier ownership problem

In this step, similar to our initial extraction step in progress 3.2.1, we employ the 'enterIdentifier' rule. However, the key distinction lies in the broader scope, where we extend beyond the declaration step to capture all identifiers and their references, including calls and callbys. Despite this expansion, we implement a refining mechanism to avoid renaming unnecessary identifiers. Additionally, we utilize the 'enterTypeIdentifier' to gather all typeIdentifiers, encompassing classes, interfaces, enums, and more. Due

to the chain renaming effect, we establish a separate renaming convention for type identifiers. It's noteworthy that in the process of renaming type identifiers, we also adjust their file names to align with the newly edited names.

There is another crucial aspect that requires thorough consideration, namely, addressing the identifier ownership problem. This problem arises due to the extensive range of possibilities resulting from the programmers' freedom of action, potentially leading us astray in all identifier-based processes. Even in the simplest codes, two identifiers can bear the same name while having entirely different origins and functionalities. To tackle this issue, we have introduced an additional criterion for each identifier: Member access.

This criterion acts as a background check for our identifiers, identifying the membership between those connected to each other by a ".", interacting with the fields, methods, and instances of other classes, creating a chained connection between them. It allows us to determine the context in which each identifier belongs and verify whether it aligns with the entities we intend to modify in our obfuscation process.To calculate these relationships, we've developed a specialized method that implements a recursive algorithm. The algorithm operates by selecting pairs of members with an overlap from the left side, returning the corresponding entity for the second element in the pair. In the next iteration, it takes the next two and queries the second element's corresponding entity given the acquired entity of the first element. This process continues until the most right member entity is resolved. With access to the corresponding entity for each individual identifier gathered earlier, we continue our process into the casting phase.

3.3.2 Casting the identifiers

In the renaming process, casting identifiers is a critical and primary stage. Although we have corresponding entities for every identifier, careful filtering and selection are necessary. Before initiating any modifications, we adhere to our selection policy to avoid renaming unwanted identifiers, such as those belonging to the 'System' class. Additionally, We maintain a list of pre-made method names from original Java classes, such as 'System,' 'Integer,' 'String,' and others. These names are carefully curated in a "should-ignore" list. Any identifiers categorized as methodDeclaration with names found in this list are intentionally excluded from the renaming process. This strategic exclusion is aimed at preventing potential issues and complications that could arise from altering these identifiers.

We continue by mapping the identifier to the corresponding entity in the table. For instance, a methodCall identifier should correspond to a methodDeclaration or interfaceCommonBodyDeclaration entity. We then conduct due diligence to ensure that both the identifier and the entity are situated within the same file and that the declaration precedes the variable section in terms of usage. In the event that these criteria are not met, we delve into potential inheritance relationships. With the establishment of type and ownership relations, the renaming process can confidently move forward.

Upon identifying a suitable entity from the table for a given identifier, the subsequent step involves changing the name of the identifier to the new designation assigned to the entity. This new name is systematically generated during the extraction process. It is paramount to observe that this generated name strictly adheres to Java's standardized naming conventions, notably commencing with a capital letter. This adherence ensures that our renaming mechanism effectively accommodates typeIdentifiers with the necessary modification of import statements.

As we advance in our endeavor to fortify data protection, the integration of complex and randomly generated identifiers introduces an additional stratum of security to our obfuscation technique.

3.4 Encryption

Data encryption is an essential security measure that involves transforming information or data into an illegible, encrypted format using cryptographic algorithms. It ensures confidentiality and security during transmission and storage, adding substantial depth to our obfuscation approach. This process unfolds in three consecutive stages:

3.4.1 Identifying encryption-required data types

Our primary focus centers on variable declarations, initialization, and assignments involving accepted primitive values such as Strings, integers, floats, and other similar data types. Within our query, a pivotal rule that guides our process is the literal rule. We meticulously employ the 'enterLiteral' rule to identify and isolate all literals within the codebase. Following this identification, we apply a series of predefined filters to ensure that these literals align seamlessly with our overarching obfuscation strategy. This approach allows us to systematically transform the code elements in a manner that enhances the security and obfuscation of the software.

3.4.2 Performing the act of encryption

We initiate the encryption process using a custom PyCipher class, which leverages standard symmetric encryption algorithms and the Python cryptography library. Each input is transformed into ciphertext, accompanied by two additional outputs: a hexadecimal Key representation and an Initialization Vector (IV); these details are recorded and available for subsequent stages in format of Python dictionary. We have established precise policies in this regard, which are categorized into two main subgroups:

3.4.2.1 Initial encryption

In this policy, we employ the pre-compile encryption method, focusing on literals and all values associated with variable declarations. These values are ideal targets as they are typically hard-coded or predefined by the programmer. Consequently, we use our PyCipher class to encrypt these selected literals.

Once we have isolated and encrypted the remaining literalContexts based on our criteria, we handle the literals connected to variable declarations differently. In this scenario, we opt for a top-to-bottom traversal approach,

Figure 8. Traversing through grammar rules: top-to-bottom

3.4.2.2 Final encryption

In contrast to our previous policy, we employ the In-compile encryption method, tailored to encrypt all value types involved in computation-intensive processes. It's important to note that in this scenario, we cannot instantly encrypt these values as we typically do because they haven't been calculated at this stage. However, we have addressed this challenge, which will be detailed in the Decryption and Parsing 3.4.3 section.

This comprehensive encryption approach covers every expression. However, it presents unique challenges, mainly when dealing with elements like member access and the variable initializer rule. Our primary focus is on assignments and equations.

A significant aspect of this process is the encryption of the right side of assignment statements. This step is pivotal in transforming complex assignments involving multiple variables and literals into a single encrypted entity value. To illustrate, an assignment statement might take "x = y + 'abc'" where the '

sign indicates an encrypted entity and "xyz" represents an encrypted literal. The result is z', a singular encrypted value stored in memory.

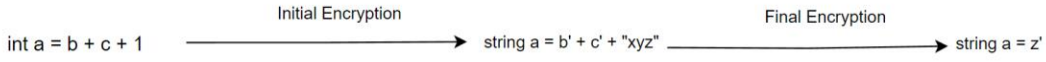


Figure 9

To maintain equality in type or return value on both sides of equality operators, we often need to convert the type or return value of the right side of the equation to match the left side. Encoding is employed as necessary to

achieve this equilibrium. It is important to note that comparing two encrypted values directly is not feasible because the encryption process yields a unique output each time, even with the same inputs. Therefore, comparisons are made by decrypting both values, enabling meaningful comparisons.

The outcome of both encryption stages invariably yields data in a String format. Consequently, we have strategically adopted the practice of systematically converting all Declarator Ids of variables into String representations. While this approach might appear unconventional, it is a deliberately intelligent choice in our obfuscation strategy. It is designed to challenge and perplex any viewer who attempts to understand the code.

By converting Declarator IDs into encrypted String values, we create a visual enigma that conceals the original meaning, thus increasing the complexity of the code. This intelligently crafted step introduces a considerable level of ambiguity into the code, making it exceptionally hard for any viewer to discern the original intent or functionality of the variables. In essence, it transforms the code into an intricate puzzle that only those privy to our decryption process can decipher.

This method is a testament to our commitment to enhancing code security while maintaining code readability for those with the proper access and decryption capabilities. By strategically obscuring the Declarator IDs in this manner, we've introduced an additional layer of complexity, rendering the code impervious to easy comprehension and analysis, thus bolstering the effectiveness of our obfuscation technique.

3.4.3 Decryption and Parsing

The encryption process, as a crucial component of our obfuscation technique, inevitably transforms the variables and assignments into a form that may seem chaotic and unreadable. To strike a balance between security and practicality, we've implemented a decryption strategy. This strategy dictates that at key moments, such as during assignments, statements, and other relevant operations, every piece of encrypted data, which may appear initially illegible, undergoes decryption. In essence, this means that whenever encrypted data is employed, it is seamlessly transformed into its original, meaningful form from our program's perspective, while appearing vague to the viewer. To achieve this goal, we need to take two essential steps:

3.4.3.1 Creating the Necessary Java Classes for Decode/Encode Functionality

The necessity of in-compile encryption and decryption leads us to create a single Java file for each input project (one for the entire project, not for each file). This Java file serves as the foundation like our

PyCipher class and resembles a hashmap named “encryptionDataMap“ that is a transformed version of our encryption data from the PyCipher’s dictionary. This Java class offers two crucial methods that we consistently utilize in this phase:

Encode:

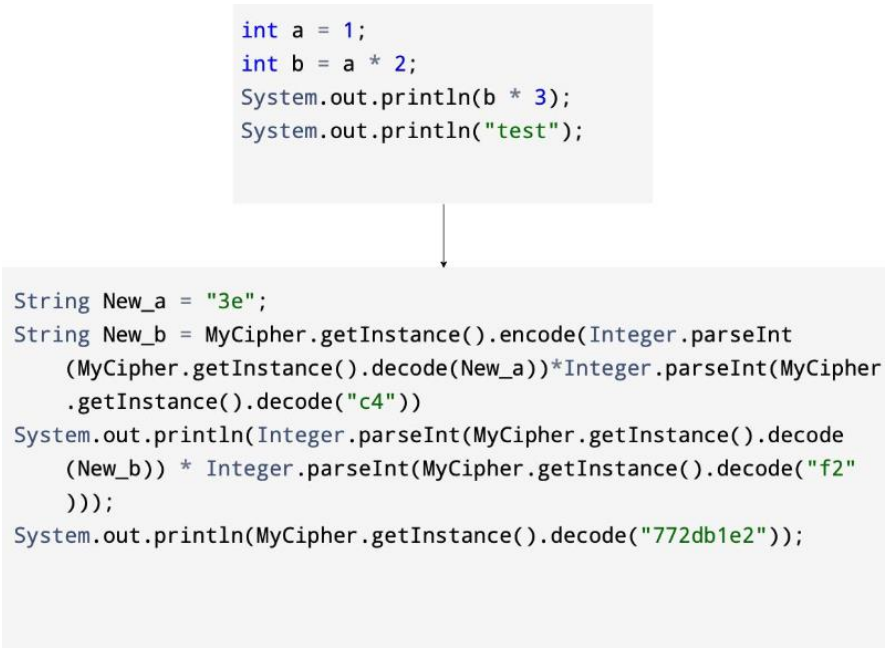
Mirrors our PyCipher class, using the Javax Cipher library to perform encryption. Following the successful execution of its mission, this method appends the standard data to our encryptionDataMap.

Decode:

For each input cipherText queries in encryptionDataMap to find the exact IV and key for the given text and then by using cryptographic algorithms converts it to it’s original form

This class will be systematically imported based on the project's packaging within all files to preempt any potential import errors.

It's crucial to emphasize that the outputs of both these methods are consistently in the format of a string. This unique characteristic necessitates an additional step in our process, where we must carefully parse these variables back into their original types. This parsing operation ensures that the variables behave correctly within their intended usage, preserving the functionality of the code while maintaining the highest level of obfuscation.



```
int a = 1;
int b = a * 2;
System.out.println(b * 3);
System.out.println("test");
```

```
String New_a = "3e";
String New_b = MyCipher.getInstance().encode(Integer.parseInt
    (MyCipher.getInstance().decode(New_a))*Integer.parseInt(MyCipher
    .getInstance().decode("c4"))
System.out.println(Integer.parseInt(MyCipher.getInstance().decode
    (New_b)) * Integer.parseInt(MyCipher.getInstance().decode("f2"
    )));
System.out.println(MyCipher.getInstance().decode("772db1e2"));
```

Figure 10. Example output of Encryption phase

3.4.3.2 Making the Final Output

In the final step of our obfuscation process, we take the output produced by the TokenStreamRewriter mentioned before for each file and create and save the new file. The new file's name is designed to match the modified class name.

To ensure this, we meticulously crafted the new files, giving them the appropriate names and incorporating the modified content based on the packaging notation we extracted from the original files. This step ensures that the resulting codebase adheres to the necessary naming conventions and structural organization, culminating in creating the obfuscated and functionally intact Java project.

The orchestration of our four fundamental phases—Noise generation, extraction, renaming, and encryption—unfolds seamlessly across various walks and parses of our Java code. This dynamic approach not only optimizes the efficiency of our obfuscation process but also facilitates concurrent execution of these phases. The parallelism inherent in their implementation enhances the overall synergy, ensuring a harmonized and intricate transformation of the codebase. This strategic concurrency aligns with our commitment to delivering a sophisticated and efficient obfuscation framework.

4 Analysis

In our analysis phase, we aimed to assess the efficiency and performance of our obfuscation technique. To conduct this evaluation, we utilized a diverse set of projects, primarily consisting of educational and university Object-Oriented Programming (OOP) Java projects. Our analysis focused on three key aspects: runtime evaluation, functionality comparison, and the similarity rate between the original and obfuscated code. This similarity rate was computed using The Levenshtein Distance algorithm.

To illustrate our obfuscation technique, we provide a simple example here. Below is an embedded code example that demonstrates the generation of the Fibonacci sequence, using 'n' as an input parameter defaulted to 10, along with its corresponding obfuscated version.

```
class GFG {

    // Function to print N Fibonacci Number
    static void Fibonacci(int N)
    {
        int num1 = 0, num2 = 1;

        int counter = 0;

        // Iterate till counter is N
        while (counter < N) {

            // Print the number
            System.out.print(num1 + " ");

            // Swap
            int num3 = num2 + num1;
            num1 = num2;
            num2 = num3;
            counter = counter + 1;
        }
    }

    // Driver Code
    public static void main(String args[])
    {
        // Given Number N
        int N = 10;

        // Function Call
        Fibonacci(N);
    }
}
```

Figure 11. Original Input Code

```java
import java.util.ArrayList;

class New_GFG {
    public static double ZSRTiyo4OBe(String NnzrSlWYs93, String[] G2JEBzuRLu2) {
        String J6VUCu50ZaW = "28";
        String AWdZhRWeKtv = "28";
        for (String GbYMP9zXb2V : G2JEBzuRLu2) {
            if (GbYMP9zXb2V.contains(MyCipher.getInstance().decode(NnzrSlWYs93))) {
                AWdZhRWeKtv = MyCipher.getInstance()
                        .encode(Integer.parseInt(MyCipher.getInstance().decode(AWdZhRWeKtv)) + GbYMP9zXb2V
                                .length());
                J6VUCu50ZaW = MyCipher.getInstance().encode(Integer.parseInt(MyCipher.getInstance().decode
                    (J6VUCu50ZaW))
                        + Integer.parseInt(MyCipher.getInstance().decode("e5")));
            }
        }
        if (Integer.parseInt(MyCipher.getInstance().decode(J6VUCu50ZaW)) == Integer
                .parseInt(MyCipher.getInstance().decode("28"))) {
            return Integer.parseInt(MyCipher.getInstance().decode("28"));
        }
        return (double) Integer.parseInt(MyCipher.getInstance().decode(AWdZhRWeKtv))
                / Integer.parseInt(MyCipher.getInstance().decode(J6VUCu50ZaW));
    }

    public static long Hm3NsyD7AZa(long BuGTtBiL3pS, char UyGaYecxOS9) {
        // Your complex logic here
        String EuLlICGrrNR = MyCipher.getInstance().encode(BuGTtBiL3pS + UyGaYecxOS9);

        return Long.parseLong(MyCipher.getInstance().decode(EuLlICGrrNR));
    }

    public static String DJ50Fr7022c(String RhKvGy74b78, boolean Jf8g44foDpY) {
        if (Jf8g44foDpY) {
            String[] HVD4akZzPCN = RhKvGy74b78.split(MyCipher.getInstance().decode("c0"));
            StringBuilder Qr36QOnzX1D = new StringBuilder();
            for (String Vgi8XmdXFV3 : HVD4akZzPCN) {
                Qr36QOnzX1D.append(new StringBuilder(Vgi8XmdXFV3).reverse()).append(" ");
            }
            return Qr36QOnzX1D.toString().trim();
        } else {
            return new StringBuilder(RhKvGy74b78).reverse().toString();
        }
    }

    public static float S6haIzMB100(double Y0yzd0642hs, long BuGTtBiL3pS) {
        String EuLlICGrrNR = MyCipher.getInstance()
                .encode((float) (Math.pow(Y0yzd0642hs, Integer.parseInt(MyCipher.getInstance().decode("b5")))
                        / Math.sqrt(BuGTtBiL3pS)));
        EuLlICGrrNR = "df3b";
        return Long.parseLong(MyCipher.getInstance().decode(EuLlICGrrNR));
    }
```

```java
static void IuMfb8biNFO(int HJSrK5gW6Hl) {

    String ScOrwOED2Ll = MyCipher.getInstance()
            .encode(-Float.parseFloat(MyCipher.getInstance().decode
                ("152f1824b0710d1e74dd9eed234e87d36218ac9a")));
    String WWf12OOxTrn = "4e59e8cd306ab752b508dad476bad45e1c5307";
    String ATCklqmbaUD = MyCipher.getInstance()
            .encode(-Integer.parseInt(MyCipher.getInstance().decode("dd8d0eb46e")));
    String H3yhWhjj55O = "28", Fsa5TuZdnWq = "e5";

    String EVoFr8zRUVP = "f5839a4c236c0c6b49ea";
    String[] Vc53iQ4KkNs = new String[Integer.parseInt(MyCipher.getInstance().decode("b5"))];
    for (String EewoOjz5IwB = "28"; Integer.parseInt(MyCipher.getInstance().decode(EewoOjz5IwB)) < Integer
            .parseInt(MyCipher.getInstance().decode("b5")); EewoOjz5IwB = MyCipher.getInstance()
                    .encode(Integer.parseInt(MyCipher.getInstance().decode(EewoOjz5IwB))
                            + Integer.parseInt(MyCipher.getInstance().decode("e5")))) {
        Vc53iQ4KkNs[Integer.parseInt(MyCipher.getInstance().decode(EewoOjz5IwB))] = MyCipher.getInstance()
                .decode("ae82d7554677");
    }
    ArrayList<String> Tiyo4OBeZSR = new ArrayList<String>();
    Tiyo4OBeZSR.add(WWf12OOxTrn);
    Tiyo4OBeZSR.add(ATCklqmbaUD);
    Tiyo4OBeZSR.add(ScOrwOED2Ll);
    ZSRTiyo4OBe(MyCipher.getInstance().decode(EVoFr8zRUVP), Vc53iQ4KkNs);
    String SgB6m0gkwOi = "28";

    while (Integer.parseInt(MyCipher.getInstance().decode(SgB6m0gkwOi)) < HJSrK5gW6Hl) {

        System.out.print(Integer.parseInt(MyCipher.getInstance().decode(H3yhWhjj55O)) + " ");

        String KLVsbzW1Dik = MyCipher.getInstance()
                .encode(Integer.parseInt(MyCipher.getInstance().decode(Fsa5TuZdnWq))
                        + Integer.parseInt(MyCipher.getInstance().decode(H3yhWhjj55O)));
        H3yhWhjj55O = MyCipher.getInstance().encode(Integer.parseInt(MyCipher.getInstance().decode
            (Fsa5TuZdnWq)));
        Fsa5TuZdnWq = MyCipher.getInstance().encode(Integer.parseInt(MyCipher.getInstance().decode
            (KLVsbzW1Dik)));
        SgB6m0gkwOi = MyCipher.getInstance().encode(Integer.parseInt(MyCipher.getInstance().decode
            (SgB6m0gkwOi))
                + Integer.parseInt(MyCipher.getInstance().decode("e5")));
    }
}


public static float Z83PSHfLWib(float IQQMOq7PEHm, short GyA2xOEjr26) {

    String EuLlICGrrNR = MyCipher.getInstance()
            .encode(IQQMOq7PEHm * GyA2xOEjr26
                    + (float) Math.pow(IQQMOq7PEHm, Integer.parseInt(MyCipher.getInstance().decode("b5")))
                    - (float) Math.pow(GyA2xOEjr26, Integer.parseInt(MyCipher.getInstance().decode("b5"))));

    return Long.parseLong(MyCipher.getInstance().decode(EuLlICGrrNR));
}
```

```
    public static void main(String HllfqIrl586[]) {

        String TKrFpUdMPFZ = MyCipher.getInstance()
                .encode(-Integer.parseInt(MyCipher.getInstance().decode("a59617e239d108")));
        String PyOMQ2HDXFX = MyCipher.getInstance()
                .encode(-Integer.parseInt(MyCipher.getInstance().decode("325ca794cacba6")));
        String LFr2dEWLXW4 = "b36dc8";
        Hm3NsyD7AZa(Long.parseLong(MyCipher.getInstance().decode(PyOMQ2HDXFX)),
                MyCipher.getInstance().decode(LFr2dEWLXW4).charAt(0));

        String RxsbWyAHFtD = "7fd025";
        String NFjv45fmUgr = "3ea4360a81b3c3dcdcc7";
        String Vv9d0paCOfW = "b395a7e66f";
        DJ5OFr7022c(MyCipher.getInstance().decode(NFjv45fmUgr),
                Boolean.parseBoolean(MyCipher.getInstance().decode(Vv9d0paCOfW)));
        ArrayList<String> COfWVv9d0Opa = new ArrayList<String>();
        COfWVv9d0Opa.add(TKrFpUdMPFZ);
        COfWVv9d0Opa.add(RxsbWyAHFtD);
        String SGjecgPwPm1 = "76f739d23c6af7b2840ad6d89f3f110ada79d8";
        String Ejb9QJeFFEi = "760bf04b584c50";
        S6haIzMB10O(Double.parseDouble(MyCipher.getInstance().decode(SGjecgPwPm1)),
                Long.parseLong(MyCipher.getInstance().decode(Ejb9QJeFFEi)));
        String HJSrK5gW6Hl = "df3b";
        IuMfb8biNFO(Integer.parseInt(MyCipher.getInstance().decode(HJSrK5gW6Hl)));
    }
}
```

Now, the generation of our analysis table for these codes is outlined as follows:

Table 2. Analysis of the output code provided in Figure 12

Version	Runtime (s)	Similarity rate	Output (for N = 10)	Correctness of output
obfuscated	+ 0.23 s	9.38 %	“0 1 1 2 3 5 8 13 21 34”	True

However, for a more in-depth exploration, you can follow this link:

http://wp.kntu.ac.ir/m.alaeiyan/Projects/Obfuscation/Data/index.html

This comprehensive analysis enables us to gauge the impact of our obfuscation on runtime performance, verify the retained functionality of the obfuscated projects, and assess the degree of similarity between the original and obfuscated code. A multifaceted evaluation is crucial to ensuring the effectiveness and reliability of our obfuscation technique in real-world scenarios.

5 Conclusion

Throughout this study, our primary focus has been on the obfuscation of entities, explicitly addressing their names, values, and, significantly, their data types. Our obfuscation process unfolds through distinct phases:

First, incorporating our sophisticated Noise generation process for heightened security, we initiate the extraction of entities and their associated properties, carefully minimizing code alterations. This includes capturing essential information such as variable names, data types, and their hierarchical relationships within the code structure. Following this, we proceed to the renaming process, where identifier adjustments take place. Finally, the encryption phase encompasses the encryption of all remaining entities, the transformation of their data types, and their parsing when utilized.

As a result, our obfuscator yields a final output: an enigmatic code where every entity features modified identifiers. Additionally, all entities with primitive data types are encrypted, converted into

strings, and dynamically parsed when employed. This comprehensive obfuscation strategy not only safeguards the integrity of the code but also enhances its security and confounds attempts at reverse engineering.

6 Future Work

Our ongoing research endeavors in code obfuscation hold promising possibilities for the future. We propose developing various encryption and decryption algorithms tailored to different data types to enhance the obfuscation process further. This diversification will enable us to accommodate a broader range of Java versions and libraries, ensuring compatibility with evolving technologies.

Efforts are also underway to optimize the runtime of the code obfuscation process. Our goal is to make it more efficient while expanding its capabilities. Future enhancements include introducing distinct renaming and encrypting styles for entities with identical identifiers. Additionally, we aim to support back-to-back obfuscations on the same codebase, further fortifying its security. With these advancements, our obfuscator will be well-prepared to seamlessly transition into the subsequent phases of CodeFlow obfuscation, abstract interpretation, and soundness theory. This holistic approach will offer a comprehensive and robust solution for safeguarding code integrity and security in the ever-evolving software development landscape.

References

[1] [Online]. Available: https://www.java.com/.

[2] [Online]. Available: https://www.antlr.org.

[3] B. Dr.D.I.George Amalarethinam, "Data Security Enhancement in Public Cloud Storage using Data Obfuscation and Steganography," in *World Congress on Computing and Communication Technologies*, 2016, 2016.

[4] M. S. Khaled M. Khan, "Data Obfuscation for Privacy and Confidentiality in Cloud Computing," in *IEEE International Conference on Software Quality, Reliability and Security Companion*, 2015.

[5] T.-W. H. Hsiang-Yang Chen, "Changing Data Type Method of Data Obfuscation on Java Software," in *Int. Computer Symposium*, Taipei, Taiwan, 2004.

[6] A. P. Shenoda Guirguis, "BronzeGate: Real-time Transactional Data Obfuscation for GoldenGate," Lausanne, Switzerland. , 2010.

[7] B. F. M. D.I. George Amalarethinam, "Confidentiality Technique for Enhancing Data Security in Public Cloud Storage using Data Obfuscation," International Science Press, 2016.

[8] [Online]. Available: https://github.com/antlr/grammars-v4.